UNLIMITED

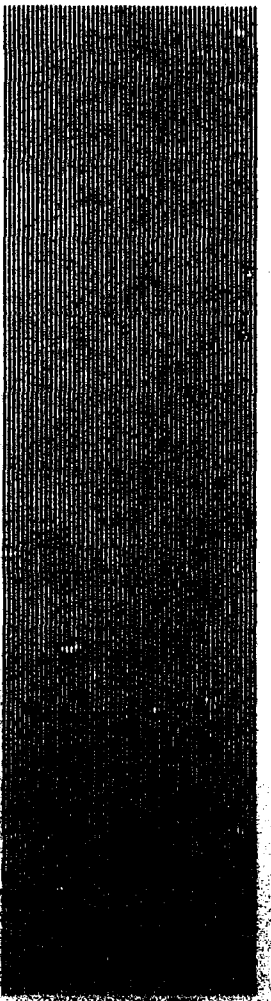
Report No. 89015

Report No. 89015

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

A STUDY OF CACHING IN CAPABILITY COMPUTERS

Author: J G Haines



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE

0057199

CONDITIONS OF RELEASE

BR-112197

****

U

****

Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISMENT

Report 89015

Title: A Study of Caching In Capabilty Computers

Author: J G Haines*

Date: September 1989

Abstract

This report describes an investigation into the use of cache memories in capability architectures. A *theory* to describe the relationship between the size of cache and its efficiency is proposed. The use of a simulator to test the theory and compare various cache management policies is described, and the results obtained are discussed. Conclusions are drawn as to the best caching strategy for the SMITE capability computer.

*Department of Electrical and Electronic Engineering
University of Newcastle Upon Tyne
Newcastle Upon Tyne NE1 7RU

CONTENTS

1. Introduction

This report describes various types of cache memory, and presents an evaluation of their efficiency. The investigation is directed towards the use of capability architectures, in particular the SMITE computer [Wiseman & Field-Richards88]. This is a capability computer based on the Flex architecture [Foster et al.82] in which procedures and abstract data types are used for security [Wiseman88]. Capability computers treat memory as a heap and this leads to a different pattern of usage compared to conventional architectures. Hence it is of particular interest to compare the effects of a cache on such systems. Only single processor systems are considered with either single or discrete caches. This is to avoid the problems of cache consistency when using multiple caches (for example in distributed systems). For a discussion of cache consistency in multiple caches [Sweazey & Smith86].

For this study, a simulator has been constructed, based upon an existing garbage collector simulator [Harrold86]. The program has been extended to include a cache simulator which enables the various types of cache to be examined, and statistics collected on their performances by running various test programs.

Refresh memory for main store usually consists of dynamic RAM's. These are small, low cost devices with low power consumption. However, these devices are slow compared with the CPU. Hence the CPU will be idle for periods of time while waiting for the memory. Faster memory devices for main store would reduce the length of the wait states of the CPU and thus increase the speed of execution. However, this is not practical, for the reasons outlined above. To increase the speed of memory accesses, but avoiding the problems associated with the faster memory, a buffer of high speed memory, called a cache, may be used. Whenever possible, the cache is used to store recently accessed data, so that it may be accessed quickly, without recourse to the slower main store.

Thus, with less time required to access data, the CPU spends a greater proportion of time performing actual processing, thereby increasing the effective processing speed. As it is expensive, it is desirable to keep the size of this high speed memory as small as possible. Other reasons for keeping the cache size to a minimum are the board size occupied by the components, and the power consumption (and hence heat dissipation) of the devices.

Performance improvements gained by increasing the size of the cache are offset by the resultant increase in cost. In light of this, the investigations performed must address the question "At what point does the increase in cost of implementing a larger cache outweigh the benefits of increased performance?"

To improve the utilisation of the cache, it is possible to manage the cache with a sophisticated algorithm, using some appropriate controlling logic. However, it is desirable to keep the complexity, and hence cost, to a minimum - unless justified by distinct performance advantages. Thus the following question should also be addressed "Is the added complexity involved in the implementation of a sophisticated cache worthwhile?"

An overview of the methods of implementing the caches used in this study is presented in section 2. This is followed by the presentation of a theory, in section 3, which attempts to predict the form of the results. A description of the simulator is given in section 4, though full documentation is available in [Haines89]. A description of the cache types investigated will be found in section 5. The test programs are described in section 6, and the results are to be found in section 7, in the form of graphs. These results are then compared with the theoretical models of cache performance, and any discrepancies arising are explained in section 8. Conclusions from the study are in section 9.

2. Design Considerations For Cache memory.

2.1 Overview.

A cache consists of a block of high-speed memory, much faster than that of the main store, in which the data that is most likely to be used again (known as **active** data) is stored [Smith82]. After the initial access to main store for the required location, further references access the cache. As a result, over a period of several references, the total referencing time for that location is reduced.

To provide this greater speed, a different technology is used. Main store memory is usually fabricated using dynamic RAM technology. Cache memory is comprised of static RAM which provides the necessary speed. In static RAM, the memory elements consist of active devices, and are hence much larger than the passive devices (capacitors) used in dynamic RAM. Hence the package size of a static RAM for a given number of elements is larger than that for the corresponding dynamic RAM device. They also have greater power consumption and, therefore, heat dissipation.

The main consideration in cache design is likely to be cost. As the fast memory required is expensive compared with that of the main store, the size of the cache must be kept to a minimum. Fortunately, to be effective, the cache need not be large due to a program's characteristics.

When a program references data, the locations are often sequential, or near to the previous location. Thus, the required locations are all contained within a small sphere of reference. This property is know as the **principle of program locality** [Coffman & Denning73] [Deitel84].

The principle of locality describes the way that programs tend to reference storage in nonuniform, highly localised patterns. It is a property which is highly likely (although not guaranteed) to occur, and is the reason paged memories are effective. In paged systems, processes tend to favour certain subsets of their pages, these pages often being adjacent. Similarly, caches may be considered in the same way as pages, but with a reduced size (often the line contains just one word).

Two forms of locality are observed; **temporal locality** and **spatial locality**. Temporal locality means that storage locations referenced recently are likely to be referenced in the near future. Spatial locality means that storage references tend to be clustered so that once a location is referenced, it is highly likely that nearby locations will be referenced. These are best shown by example. If the sun is shining at mid-day, then it is highly probable that the sun was shining at 11:30, and that it will still be shining at 12:30. This is an example of temporal locality. Spatial locality may be described thus: If the sun is shining in Newcastle then it is likely that the sun is also shining in Cullercoats and Whitley Bay.

These forms of locality arise from the way programs are written and data is organised. Observations of temporal locality are supported by looping, subroutines, stacks, and variables used for counting and totaling. Observations for Spatial locality are array traversals, sequential code execution, and the tendency of programmers to place related variable definitions near one another.

As described previously, a program will tend to reference the same locations several times before abandoning them for other data. Storing these locations in the cache will, therefore, result in an increase in speed. This phenomenon is demonstrated clearly by the example shown in Figure 2.1. In this example of temporal locality, the location used to store i is accessed several times for each execution of the loop. Hence the data can be cached usefully. The code of the loop is executed UPB v times. This is an example of Spatial locality and the loop's code may be usefully cached.

```
FOR i TO UPB v
DO
   v[ i ] := v[ i ] + 1
OD;
```

Figure 2.1: Example Of Locality Of Reference.

2.2 Measuring Cache Efficiency.

To enable a comparison of efficiency to be made, two parameters involved in the cache operation must be studied. The two parameters used are **hit rate** and **update rate**. Hit rate is a measure of the number of times the required data may be found/stored in the cache as opposed to main store.

When data is inserted into the cache it displaces the old data. If the data in main store is inconsistent with the displaced data, the main store must be updated or else the value will be lost. Thus to obtain high efficiency, the number of these updates should be minimised - hence a low update rate is preferable.

2.3 Cache Structure.

The mapping of a main store location onto a cache location may be arranged in one of three ways: direct mapping, set-associative, or fully associative.

2.3.1 Direct Mapping Cache.

In a direct mapped cache each primary location is mapped onto a single position in the cache. To effect this, some nominated number of low order bits of the address are used to provide an **index** (also known as offset) within the cache. Each index contains a **line** (see section 2.5) consisting of one **entry** (a single item of data - usually a word). The size of the cache dictates the actual number of bits used to form the index. The rest of the address (the **tag**[1]) is stored in the cache alongside the data (Figure 2.2).

---

[1] In this report, the word tag is exclusively used to describe the high order bits of a memory address used to deduce whether a cache entry is the desired entry. It should not be confused with the tag bit in the SMITE computer, which is used to differentiate between pointers and data.

On a read request from the CPU, the index into the cache will be determined by the lower order address bits, and the tag value at this index is compared with the high order bits of the address to determine if the cache is holding the value of the location requested. If the tag matches, the request is a 'Hit', and the data is taken from the cache. If the tag is not equal, the request is a miss, and the data is obtained from the primary memory and the cache updated so that further requests for the same location are found in the cache. To retrieve the data from main store and update the cache, one of the policies described in section 2.6 is applied.

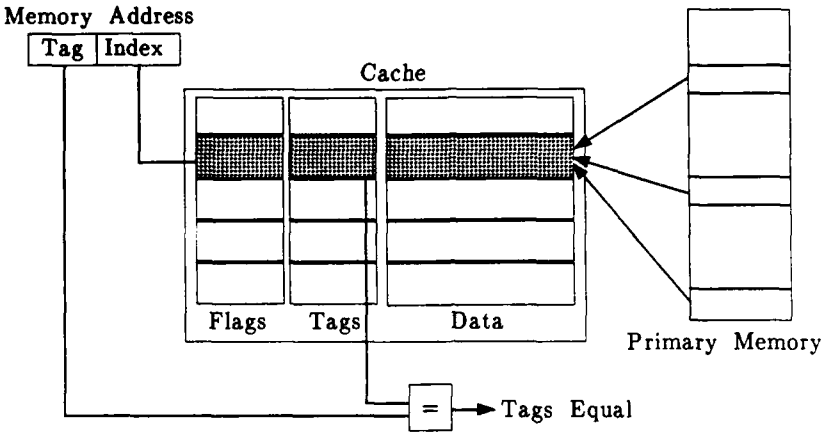


Figure 2.2 Direct Mapping cache.

In order to perform some housekeeping duties, other bits are also incorporated into the structure. The '**valid**' bit is used to ensure that only valid data is taken from the cache. The '**consistent**' bit, is only used when the writeback policy is employed. It shows whether or not the data held in the main store is consistent with the cache, and hence if the main store should be updated when the cache entry is replaced with data from a different location.

This structure has the advantage of being inexpensive to build, with no complex hardware required, and a simple cache maintenance policy.

However, this structure has a drawback. Each index into the cache will have many primary memory locations associated with it. Whenever a reference is made to a specific index, if the request is a miss, the data must be retrieved from the main store, and the old contents of the index discarded to make way for the new data. This may cause active data to be discarded prematurely, thereby reducing the effectiveness of the cache.

The only policy available to control this structure is the direct displacement algorithm, described in Section 2.6.

3.2 Fully Associative Cache.

In a fully associative cache, the contents of a primary location may be stored in any line of the the cache (Figure 2.3). To identify which location is stored in a particular line, the entire primary address is stored as the tag alongside the data. When the CPU accesses memory, all of the tags within the index must be compared until either a hit is obtained, or all tags have been checked against the CPU address. To maintain the speed advantages of the cache, all tag comparisons must be performed in parallel. This increases the complexity of the cache, and can be very expensive in terms of hardware. More physical memory is required as the tags hold the full address, not just some low order bits as in the direct mapping cache. It does have the advantage, however, that active cache entries are less likely to be discarded. Thus, with less main store accesses to store the displaced data, the fully associative cache should be more efficient than a direct mapping cache. Any of the cache policies described in this section may be used to control this structure, except for the direct displacement policy described in section 2.6.

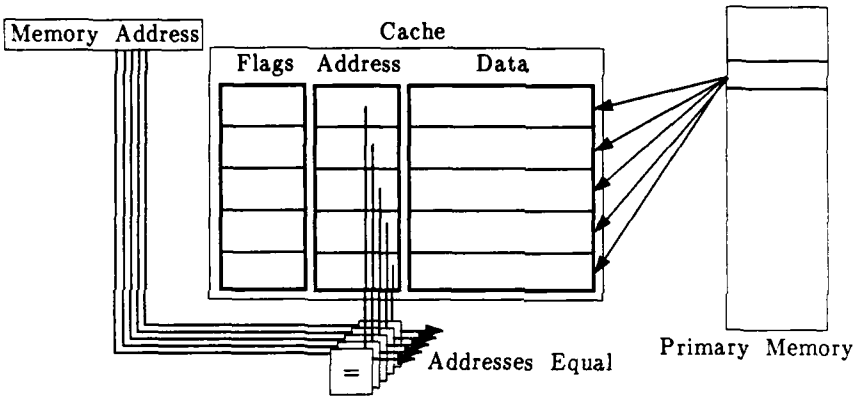


Figure 2.3 Fully Associative Cache Mapping.

2.3.3 Set-associative Cache.

The set-associative cache is an attempt to reduce the complexity of the fully associative cache whilst still retaining the ability to retain active data longer than a direct mapping cache. As used in a two-way form on the VAX 11/780, the set associative cache is a compromise between the simplicity of the direct mapping structure and the more efficient, complex displacement policies of the fully associative cache.

In an n-way associative cache there are n lines for each index in the cache. Thus, when a CPU request is made, only n checks have to be made for a tag match, the set of n which are checked being determined by the cache index specified by some number of lower bits of the address, as used in the direct mapping cache. Thus, compared with a fully associative cache, the number of parallel checks required is reduced, as is the likelihood of deleting active data from the cache.

As for a fully associative cache, all policies described may be used to control this structure except the direct displacement policy.

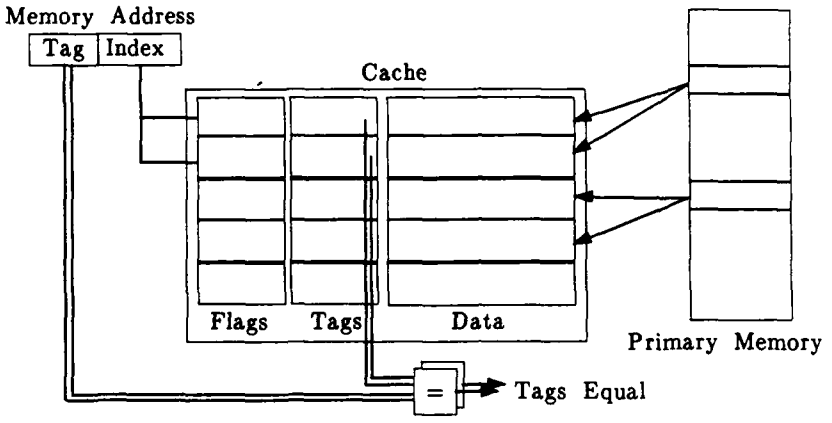


Figure 2.4 Set Associative Cache Mapping (2-way)

2.4 Caching By Data Type - Discrete caches.

Within a conventional computer architecture, code and data usually occupy disjoint sections of memory, and so it is possible to differentiate between them. In a capability computer, this idea is extended and memory is organised into a variety of types of block, e.g code, data, procedure, stack. Due to the way blocks are accessed, the block's type is always known. This information can also be used to select from a number of discrete caches, so that each caches data from a particular type of block. This should provide the benefits of the set-associative cache while keeping the simplicity of the direct mapped cache.

2.5 Line Widths of a Cache.

All of the cache structures described contain one word of data per line. However, it is possible to have more than one word per line. The line consists of the desired location, followed by n-1 subsequent locations in memory. The line width will then be n. Thus, sequential words of data will be found in the same line of the cache. The structure is otherwise identical in operation to those already described, except that extra decoding of the address must be performed to create the index. Performance will thus be related to line width. This is due to the nature of the Spatial locality of the program, particularly that of the code. Thus subsequent accesses are likely to be found in the same line as the previous access. However, the resulting cache will be considerably more complex since logic is required to move wide cache lines to and from main store using a narrow data bus.

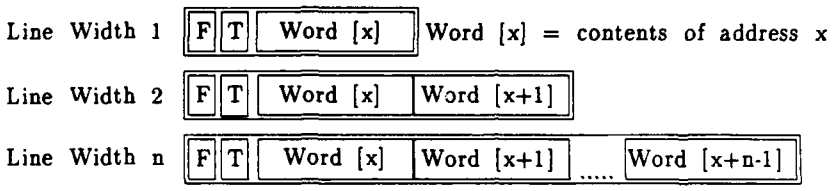


Figure 2.5 Examples Of Line Widths.

2.6 Displacement Policies.

Due to constraints imposed by the structure of the cache, some method of displacing data already in the cache is required to enable new data to be stored. This section describes the various displacement policies which have been developed.

2.6.1 Policy 1: Direct Displacement Algorithm.

The index of the entry concerned is given by some low order bits of the address of the data. If a hit occurs the data is read from, or written to, the appropriate index in the cache. On a read miss, the data is fetched from main store, and inserted in the appropriate index, thereby overwriting the current entry. If the writeback scheme is used the old data must be written to main store before the new data can be written to the cache, to maintain consistency. A flag shows if the cache entry is consistent with main store, thereby avoiding unnecessary writes to main store. On a write miss, the data is written to the appropriate index in the cache, after updating the main store if necessary (writeback).

2.6.2 Policy 2: Random Displacement Algorithm.

The thinking behind this policy is that fixed policies work well for certain programs, but badly for others. By randomly selecting the entry to be displaced, no particular program is favoured.

On a read access, if the read is a miss, the cache must be updated from main store. The entry to be overwritten is determined by the result from a random number generator. The randomness of the generator should be sufficient to ensure that each entry has as long a lifespan as possible before being deleted to make way for another entry, thus no bias is given to any particular entry. This policy is the simplest available to control associative caches. It is also useful as a benchmark with which to compare the other policies.

2.6.3 Policy 3: Least Recently Used Displacement Algorithm (LRU).

This policy is concerned with the number of times a cache entry has been used. It is an attempt to reduce the number of deletions of active data by marking the usage of each entry. When an entry is to be deleted, the algorithm selects that entry which has been the least recently used of all entries in the cache. This is based on the principle that the more recently used entries are more likely to be requested again in the near future (temporal locality). Hence the policy is dependent upon read requests. Write requests may be labelled in one of two ways. The first option is to label them most recently used. This is because they are expected to exhibit temporal locality, and be required in the near future. Alternatively, the entries may be labelled as least recently used, as the structure of programs is such that data is read, altered, and then written. Hence the data will not be used in the near future. Labelling write requests as most recently used is most promising, as data does not always undergo a read/amend/write cycle, but is likely to exhibit temporal locality (including read/amend/write cycle).

2.6.4 Policy 4: Least Recently Cached Displacement Algorithm (LRC).

This policy marks the order in which entries are written to the cache. The algorithm then selects the least recently cached data to be deleted. Read requests can be treated in one of three ways. A read request may be labelled least recently cached, as once read, it is unlikely to be read again (due to the read-alter-write cycle). It may become labelled most recently cached, as it is likely to be accessed in the near future. It is also possible to leave the label unchanged, as if the read-alter-write cycle occurs, it will be updated on the write cycle.

2.7 General Management Schemes.

Whichever displacement policy is used, and whatever cache structure is implemented there are two methods of updating the store: **writethrough** and **writeback**.

2.7.1 Writethrough.

Every time a CPU write request occurs, the main store is updated, irrespective of whether the request is a hit or a miss. This has the advantage that the main store is always consistent, of particular importance in systems with multiple processors sharing the same memory. It should be noted, however, that writethrough does not guarantee cache consistency in multiprocessor systems [Sweazey & Smith86]. If writethrough is used, the speed advantages of the cache will then only be apparent on read requests, unless main store accesses are buffered or performed in parallel with inserting the entry into the cache.

In some applications the bus used for main store accesses may be used for DMA accesses. In this case a writethrough policy will cause delays; this is because the cache updates can conflict with the DMA accesses. The writethrough scheme is wasteful if the data is modified after having been written, and also gives rise to problems if the cache is more than twice as fast as main store. If a CPU request is a cache miss, a delay will occur until the writethrough has completed and main store can be accessed again.

2.7.2 Writeback.

The writeback scheme aims to avoid the possible speed disadvantages of the writethrough scheme by updating main store only when required. That is, only when the data stored in the cache is required to be deleted to make way for active data. It has the advantage that the number of writes to memory is decreased, and hence the speed advantages of the cache are more fully utilised, but with the drawback that the main store is not consistent. The system may be optimised by using the spare cache/main memory cycles for the updating of the main store thereby reducing CPU waits even further.

3. Theory.

3.1 The Effect of Increasing Cache Size.

An efficient policy will score a high hit rate for both read and write requests, although one may be higher than the other. However, the characteristics of both will be similar, as explained later. For a writeback scheme, update rates should be low because updates can only occur on a cache miss, and even then are often unneccessary. Idealised curves representing hit and update rates against cache size are shown in Figure 3.1 and Figure 3.2.

The hit rate curve can be seen to have four distinct regions. The first, the **toe**, is expected to be a region of erratic results. This is due to the fact that small changes in cache size represent a large proportion of the cache itself. The second region is that which is expected to be observed in most practical systems. This may be described as the **practical operating region**. In this region, as the cache size increases so the rate of increase in performance decreases. However, for practical purposes, it may be seen to be approximately linear up until the **knee** of the curve.

Saturation occurs for the following reason: the program being run does not use all the primary memory, and the locations used are often split. When the size of a cache is increased, the locations mapped onto a particular entry may consist entirely of the locations unused by the program. Thus, as the locations are never accessed, neither are the cache entries, and so the efficiency remains the same. Continuing this still further, a point is then reached when the size of the cache encompasses all the store required by any process (all the block types used). Thus each location required is mapped onto a unique cache entry. The rate then jumps to 100%, whereupon increasing cache size becomes pointless. This idealised view is supported by results obtained by [Wong and Morris88].

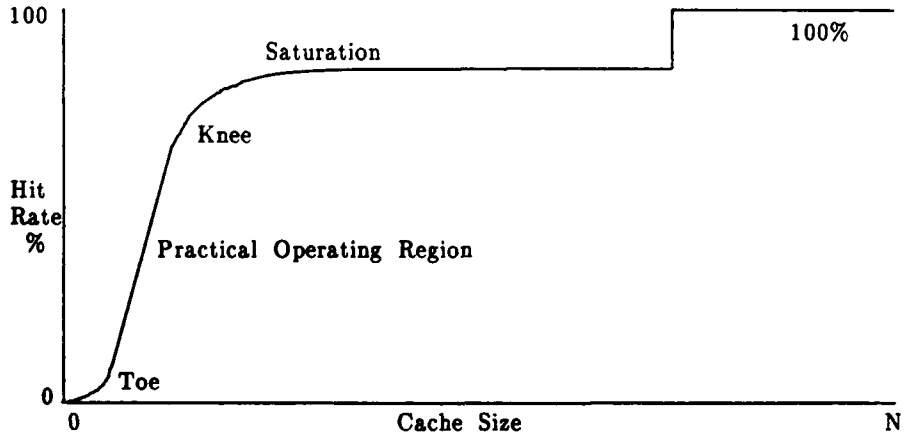


Figure 3.1 Idealised Performance Curve for Hit Rate.

In the case of updates, the situation is considerably more complex. The number of updates depends not only upon the number of hits (which would result in a graph the inverse of that for hit rate), but also the volatility of the data used by the program. An update only occurs on a miss, and only if the data held in the cache is inconsistent with that of main store. Therefore, the update rate will be less than, or equal to the miss rate. As the memory is not necessarily updated on a miss, the practical situation will be unique for each process which is executed. Thus the results may not bear any recognisable resemblance to the idealised case.

The ideal case initially remains constant for a short period, then shows a sharp decrease, following an exponential decay curve. This effect is seen because, for small cache sizes, a small increase in cache size will have quite a profound effect on the hit (and, therefore, miss) rate. Not only will this effect be present, but as the cache size increases, the amount of time for which a location remains mapped onto a cache entry is increased. Thus, the possibilty of the data being amended whilst in the cache is greater, thereby increasing the chance that the main store will have to be updated when the location is finally displaced from the cache.

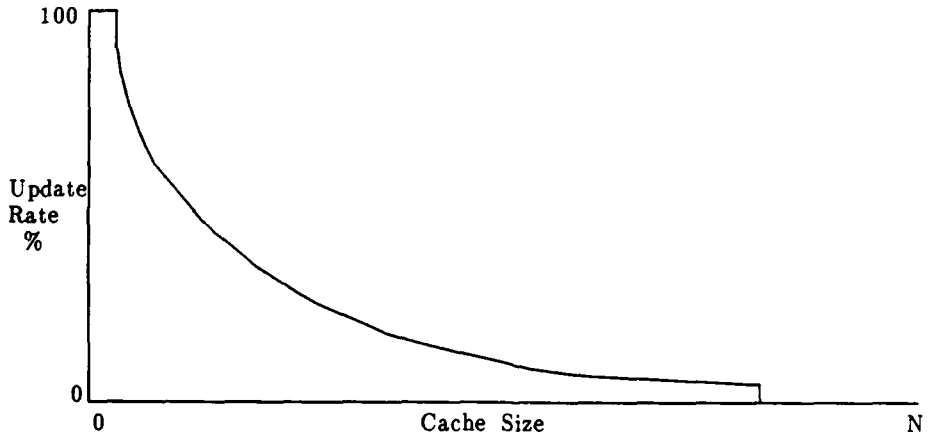


Figure 3.2 Idealised Performance Curve For Update Rate.

3.2 The Effect of Line Width.

As described by the principle of spatial program locality, memory accesses frequently operate on small blocks of data. The use of a line width greater than one (for example 8 or 16) increases the hit rate in the following way. The line will be 'primed' by the first of a sequence of accesses. Any accesses thereafter are likely to be in the line, and therefore be hits. Thus, the larger the line width, the greater the probability of hits. The degree of increased hits is, of course, dependant on both the size of the line, and the locality of the data involved. However, there is a penalty associated with line widths greater than one. Several reads are required to fill a line which is wasteful if they are unused. For the writeback scheme, several writes will be required every time the main store needs to be updated, or else complex logic is required to operate on partially filled lines.

3.3 The Effect of Changing the Degree of Associativeness.

The graph of performance against **degree of associativeness** (an N-way associative cache has N degrees of associativeness) is expected to have an asymmetric curve similar to that shown in Figure 3.3. For a given cache size, the direct mapping cache will have the poorest performance. However, it should be noted that after reaching a peak, the perfomance decreases until the cache becomes fully associative. The fully associative cache is not as efficient as some other forms of cache, but is more efficient than the direct mapping cache (see section 2.3).

The direct mapping cache discards data due to the cyclic nature of the cache. For example, in a ten element cache, the eleventh entry will displace the first entry, the twelfth the second entry and so on. This effect is most pronounced in a direct mapping cache, and less so for the associative caches. In a similar manner, the fully associative cache discards data due to the cyclic nature of the displacement policy. Take, for example, a ten way cache; the eleventh unique access will discard the entry with least priority, the twelfth will discard the entry with the next lowest priority, and so on.

The improvements seen in the set associative caches are due to the fact that active data is less likely to be discarded, as any new data may be stored in one of several locations. It could be argued, by the same reasoning, that a fully associative cache should, therefore, be the most efficient structure available. However, the cyclic nature of the displacement policy of the fully associative cache reduces the hit rate, although it is not as prominent as the characteristics of a direct mapping cache.

The peak occurs at a point slightly before that where the degree of associativeness is equal to the number of lines in the cache. The peak is due to the fact that two effects described above are both subdued (the effects reach a maximum at the structural extremes). The offset nature of the curve is due to the differing strengths of the two effects described.

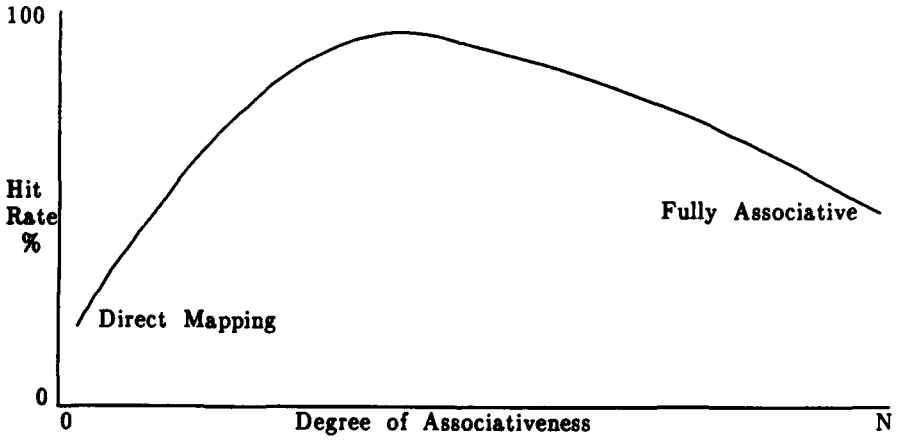


Figure 3.3 Idealised Cache Performance vs Degree Of Associativeness.

3.4 The Effect of Different Policies.

The results from the random policy cannot be readily predicted, but as they appear dependent upon chance, they provide a suitable benchmark with which to compare the other policies. Thus efficient policies will yield better results for both hit and update rates.

The results obtained from using a direct mapping cache are expected to be similar to that of the two-way set associative cache using a random policy. This is because the 2-way associative cache should provide a similar performance to the direct mapping cache, as the associativeness compensates for the reduced number of indexes, but does not yield any noticeable gains in hit rate. This is due to the use of the random policy, which does not make full use of the associative structure. Hence, the use of the random policy will not yield the improvements expected from the LRU and LRC policies.

The use of LRU and LRC policies with the two-way associative cache should show a slight improvement over the other two methods. This is because the policies mentioned should filter the entries, discarding the most redundant data in preference to active data. This should enable more active data to be stored in a given cache size, thereby increasing the efficiency.

Overall, the results should be similar. Although variations due to the type of cache used will be apparent, they should be minimal. A set of much larger test programs should clarify any differences.

3.5 The Effect Of Discrete Caches.

In a conventional architecture, a single cache is used, in which all locations used by the program are cached indiscriminately; no distinction is made between types of data. It would be possible to use a memory management unit to split the main store, keeping code and data separate. This would allow two caches to be used, however, the consistency of the caches cannot be guaranteed, since code and data may overlap

In a single cache, code or data may be stored in the same entry in the cache. Thus **interference** may occur, whereby caching a data word may delete a code word and, similarly, caching a code word can delete a data word. Execution of a program will, typically, comprise the reading of an instruction (code), amending a memory location (data) and then reading the next instruction (code) and so on. This presents ample opportunity for interference to occur, which will reduce the performance of the cache.

If the code and data could be segregated, deletions due to interference would be avoided, and the overall performance of the cache increased. In the case of capability computers, not only are data and code disjoint, but some distinction is also made between other types of data. Thus, it is possible to implement several discrete caches. The data type of a block is always known, and this can be used to specify the cache to be used. Hence it should be possible to use several smaller caches, with a total size of that of a single cache, which together yield a greater overall performance than that of a single cache. Alternatively, smaller caches could be used, whilst providing the same level of performance.

4. The Simulator.

The simulator is an extension of the garbage collection simulator documented in [Harrold86], and with very little modification enables a study of either incremental garbage collectors, cache policies, or both. The new program is fully documented [Haines89].

The program simulates a simple instruction set based upon Flex [Foster et al.79]. This includes first class procedures [Foster et al.82] [Currie et al.81] implemented as closures.

The simulations do not take into account the multiprogramming normally carried out by many computers. In such systems, many processes are executed 'simultaneously'. It is not possible to determine the effect this may have using the simulator as it can only simulate one process at a time.

It should be noted that the associative mapping schemes have line widths of 1. This is somewhat different to the scheme used in many practical cache designs, which use a line width up to about 16 (for example, the IBM 3033). This simpler scheme is necessary due to the length of time to simulate wide lines, as the simulator, unlike the hardware cannot enact upon lines in parallel, but processes elements sequentially.

5. The Algorithms Under Investigation.

Each program is simulated using one of four options, making full use of the facilities of the simulator, and providing a comparison of several policies. The options used are:

1) Direct mapping.

2) 2-way Associative mapping, with Random (policy 1).

3) 2-way Associative mapping, with Least Recently Used (policy 2).

4) 2-way Associative mapping, with Least Recently Cached (policy 3).

The degree of associativeness has been investigated for program two, although not using all options available on the simulator. This can not be investigated fully as the simulator takes an inordinate length of time due to the serial checking of tags (an unavoidable delay). Although not a thorough investigation, the results obtained should be representative, and enable a comparison with idealised results to be made.

The effect of using several caches to store specific types of data was also investigated, using only the direct mapping policy, and compared with previous results.

6. Description of Test Programs.

Testing is performed on several programs, each operating on a file of test data. All conditions are kept constant between runs of the same program. In addition, the results obtained are for '**cold start**' conditions, i.e all cache entries are invalidated (or 'cleared') before running the test program.

Four programs are described here, and are representative of programs commonly used, although they do not neccessarily represent the best suite of test programs.

Test Program 1: Towers of Hanoi.

This program provides a solution to the Towers of Hanoi problem for a specified number of 'discs'. The algorithm used for the solution is recursive.

Only the solution for three discs was obtained, as this represented the smallest meaningful value for the problem. Larger values were not chosen, due to the time taken to run the program. As the program is recursive, the results are expected to be independant of the number of discs.

Test Program 2: Word.

This program takes some text, delimited by *, and strips off the blanks. Each line is cut into a word, leaving the rest of the line. The word is then stored in a block, whose size is the number of characters in the word. This is repeated until the end of the line, and then for all lines.

An extract of a report was used, representing a typical text (whatever that may be). The data is long, and hence takes hours to process. To reduce computing time, the text could possibly be shortened and still provide meaningful results.

Test Program 3: Wordsort.

This program sorts a specified number of words into alphabetical order, using the bubblesort algorithm.

A random selection of words was used, being taken at random from "A Guide To American Football", and a selection of people's names (which are not quite random, but are arranged randomly).

Test Program 4: Frag.

This program is similar to program two, but to increase fragmentation, a list of words beginning with certain letters is kept by loading a pointer to the workspace.

The same data as used for word was also used here, again for similar reasons, and again it should be possible to reduce the size of the text without invalidating any results.

7. Results.

7.1 Single Cache.

In a single cache, words are stored irrespective of the block type.

The results shown in Figures 7.1-7.4 are from running a recursively structured program which gives a solution to the Towers of Hanoi problem for three discs.

Figure 7.1. Read hit rate against cache size for program one.

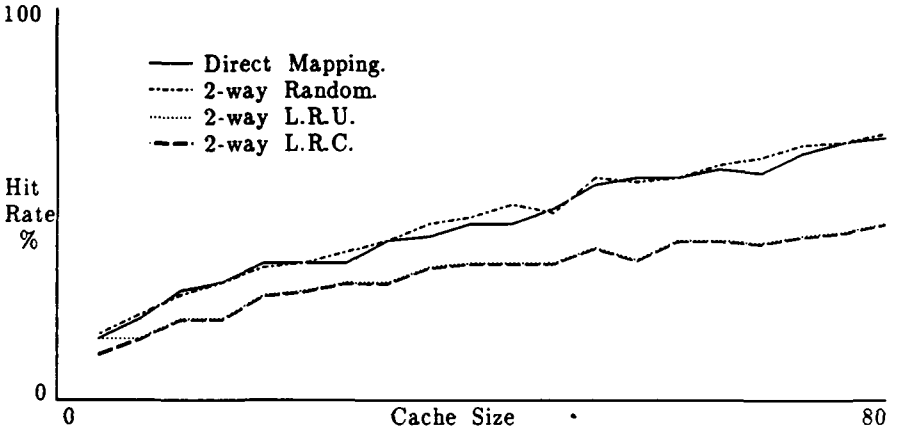


Figure 7.2. Write hit rate against cache size for program one.

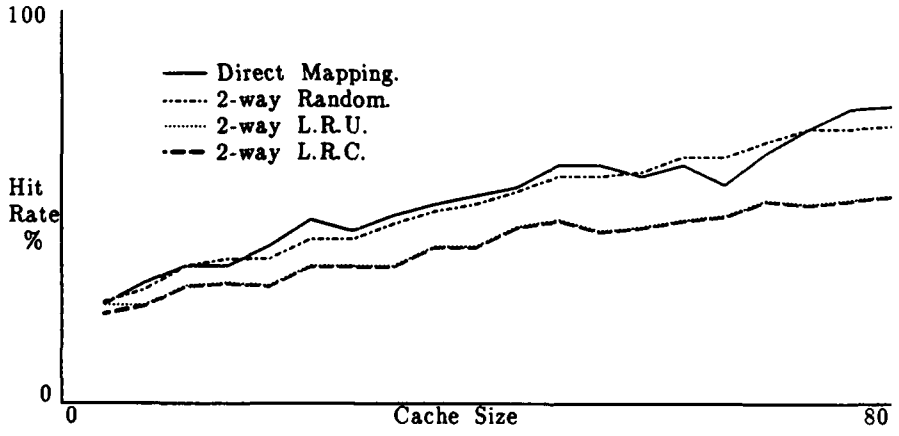


Figure 7.3. Read update rate against cache size for program one.

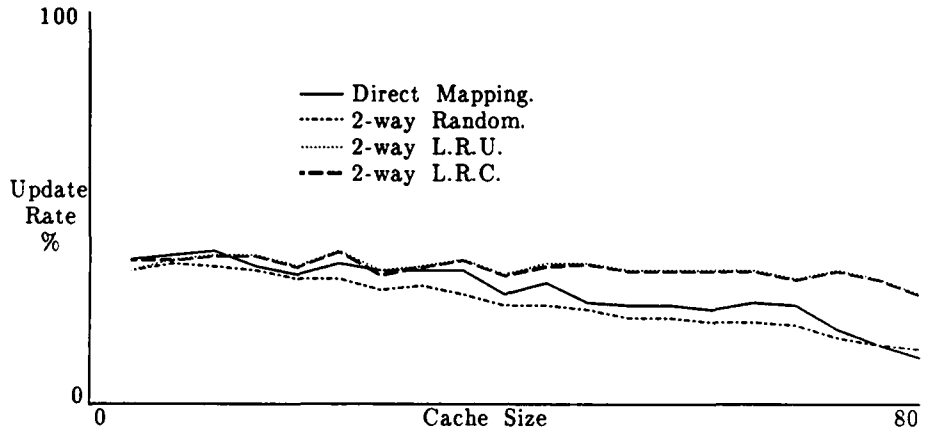
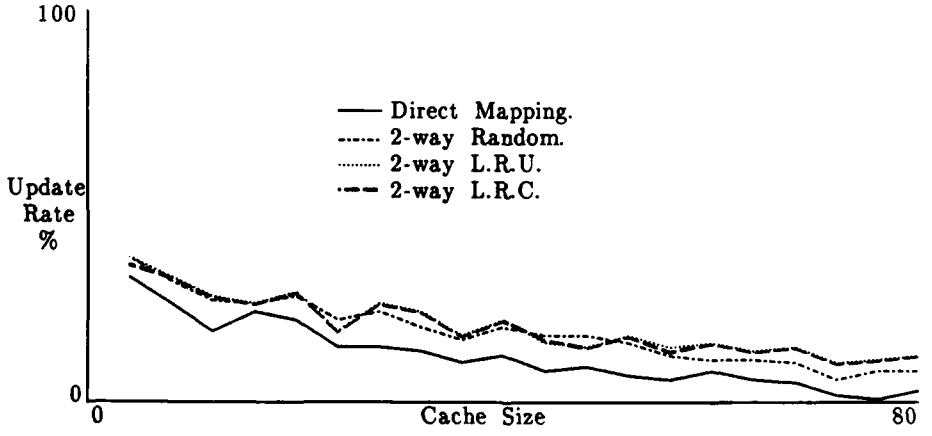


Figure 7.4. Write update rate against cache size for program one.



The results shown in Figures 7.5-7.8 are from running a program which strips the blanks from a text, and creates blocks containing words.

Figure 7.5. Read hit rate against cache size for program two.

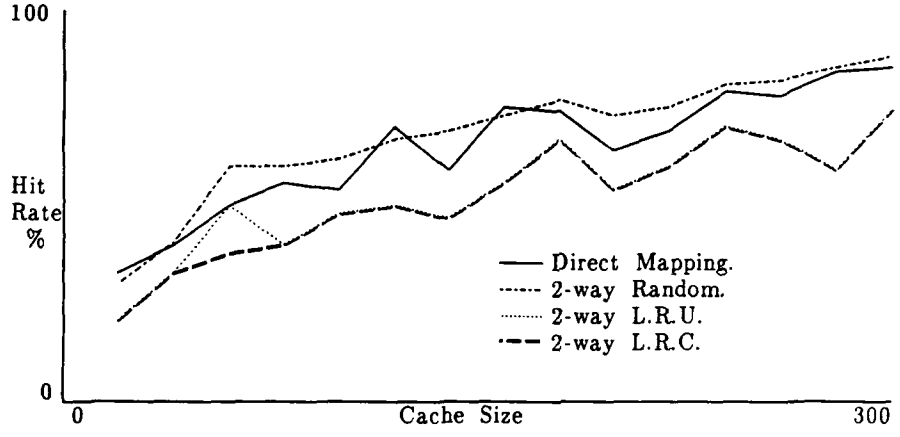


Figure 7.6. Write hit rate against cache size for program two.

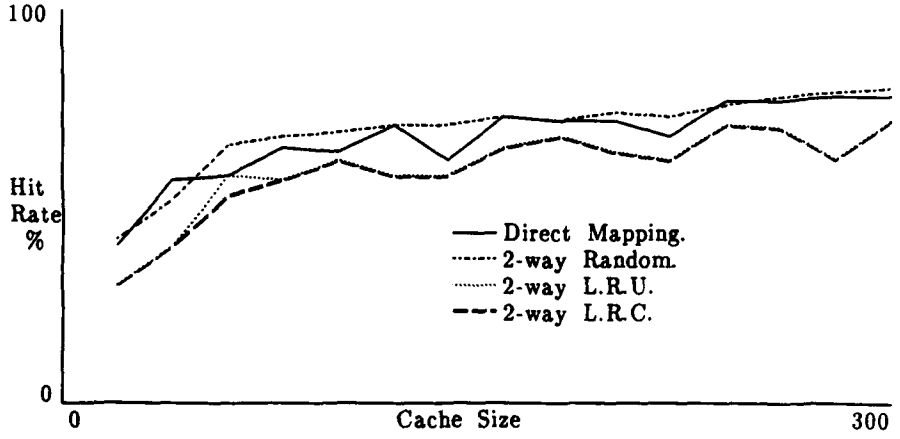


Figure 7.7. Read update rate against cache size for program two.

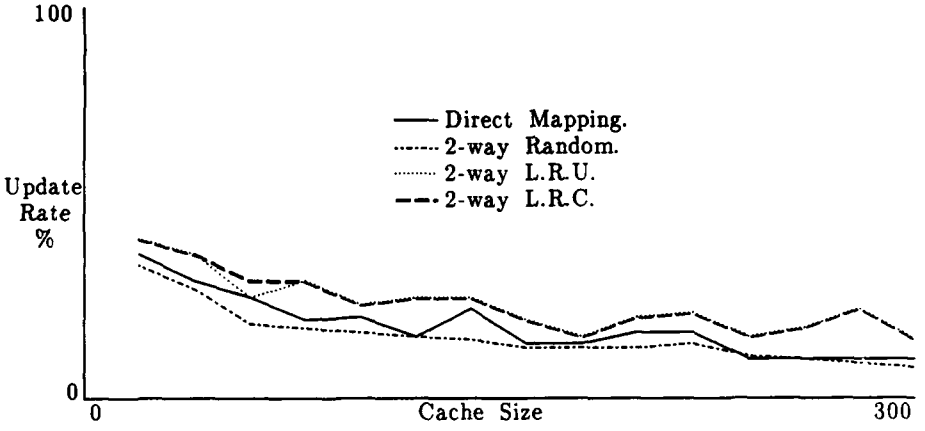
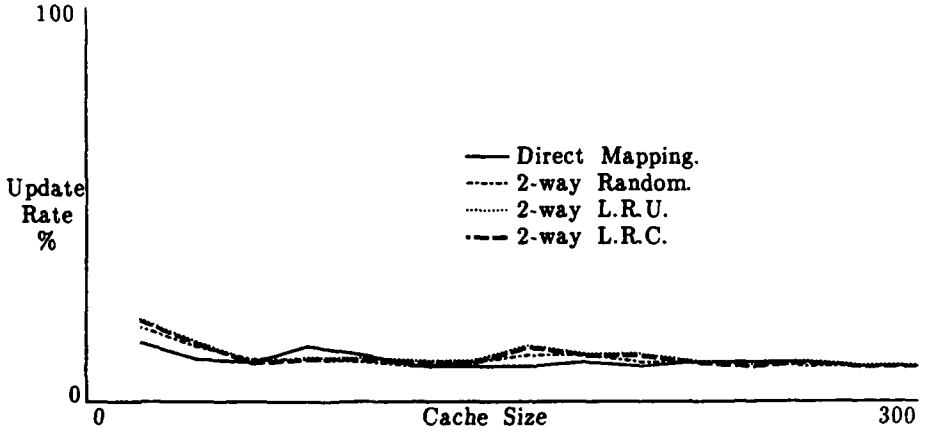


Figure 7.8. Write update rate against cache size for program two.



The results shown in Figures 7.9-7.12 are from running a program which sorts a list of words into alphabetical order.

Figure 7.9. Read hit rate against cache size for program three.

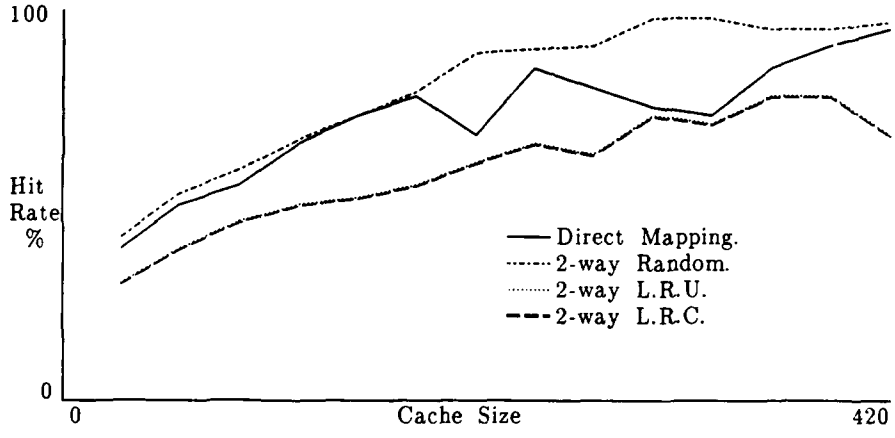


Figure 7.10. Write hit rate against cache size for program three.

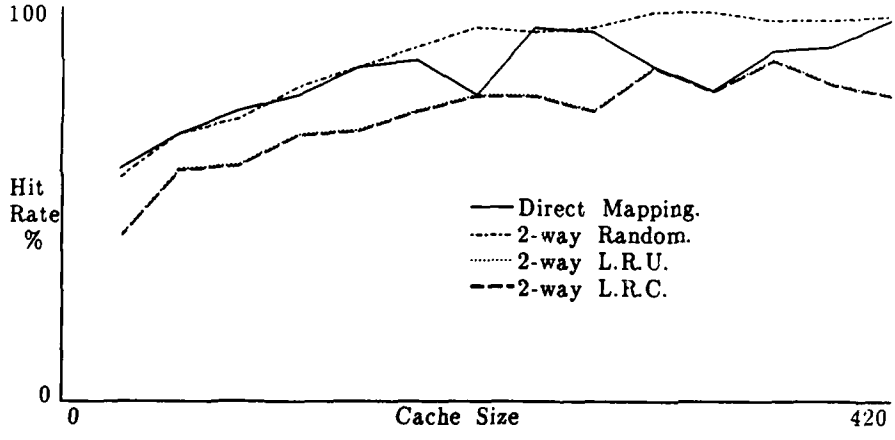


Figure 7.11. Read update rate against cache size for program three.

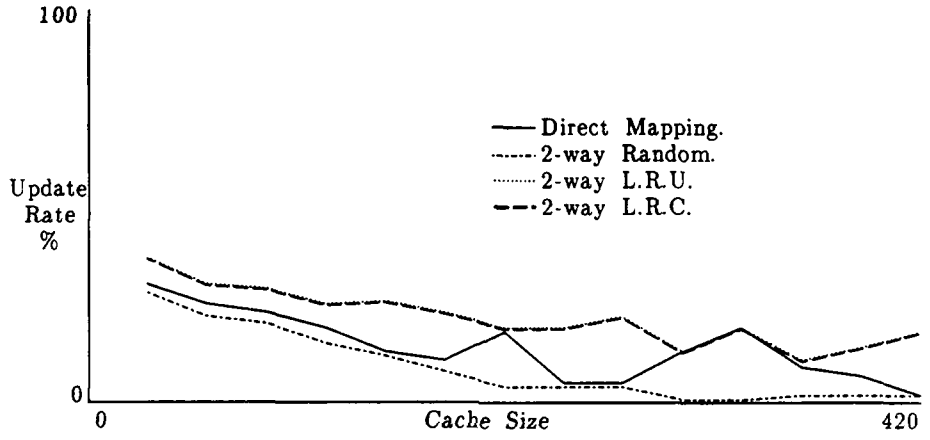
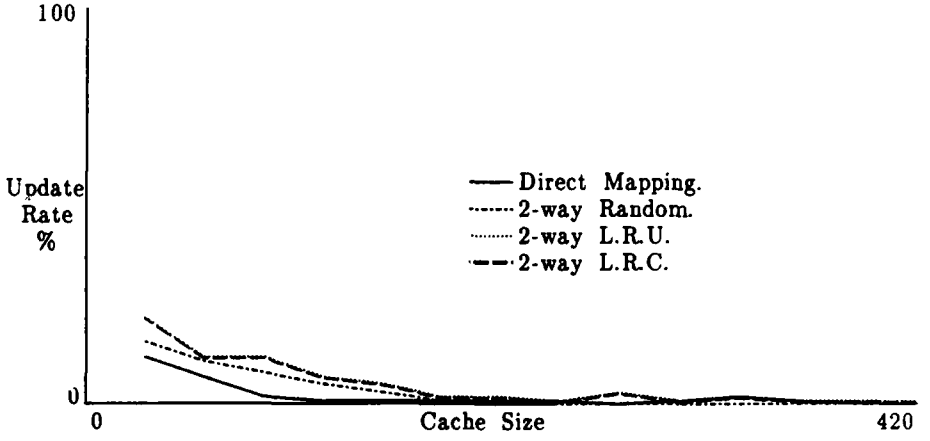


Figure 7.12. Write update rate against cache size for program three.



The results shown in Figures 7.13-7.16 are from running a program which strips the blanks from a text, creates blocks containing words, and keeps a list of the stored blocks. .

Figure 7.13. Read hit rate against cache size for program four.

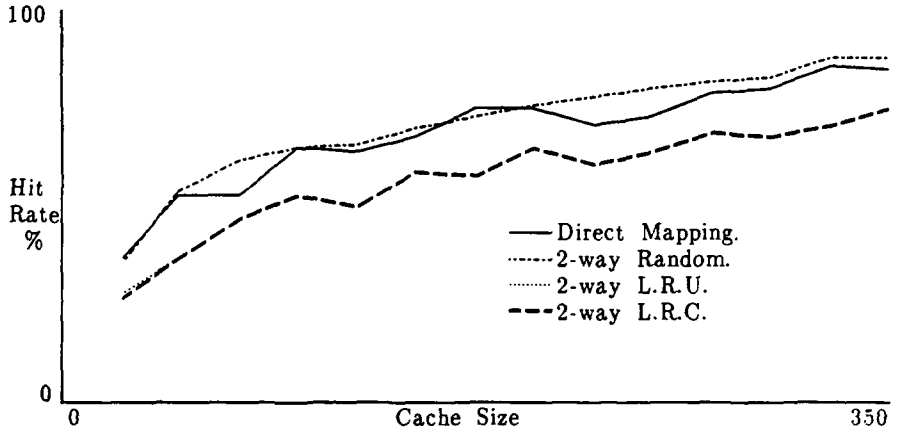


Figure 7.14. Write hit rate against cache size for program four.

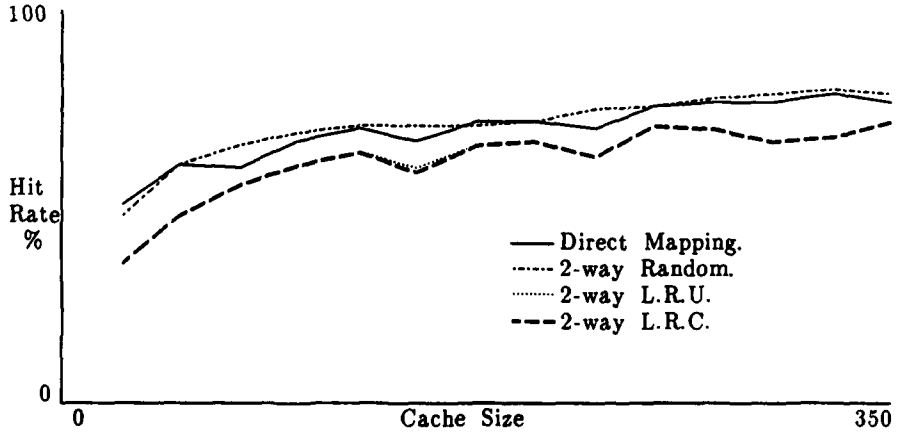


Figure 7.15. Read update rate against cache size for program four.

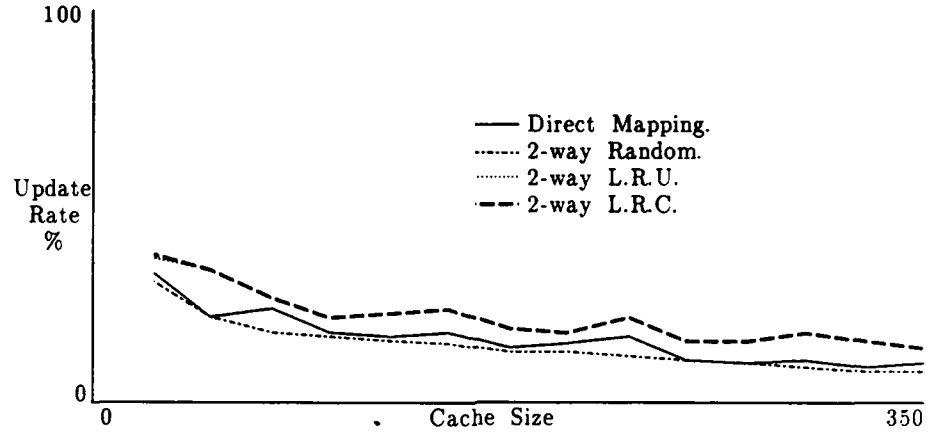
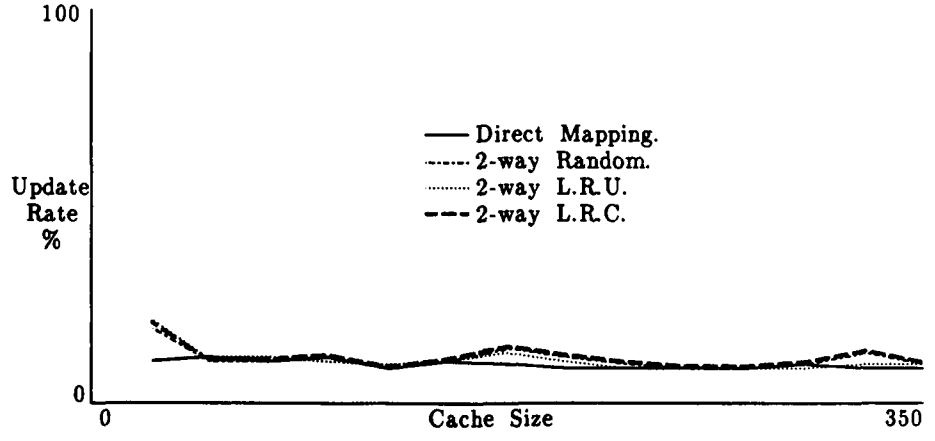


Figure 7.16. Write update rate against cache size for program four.



7.2 Discrete Caches.

Discrete caches are several caches, each storing a particular type of data. The caches used here store workspace, code, closure, and non-locals.

The results shown in Figures 7.17-7.20 are from running a recursive program which gives a solution to the Towers of Hanoi problem for three discs.

Figure 7.17. Read hit rate against cache size for program one.

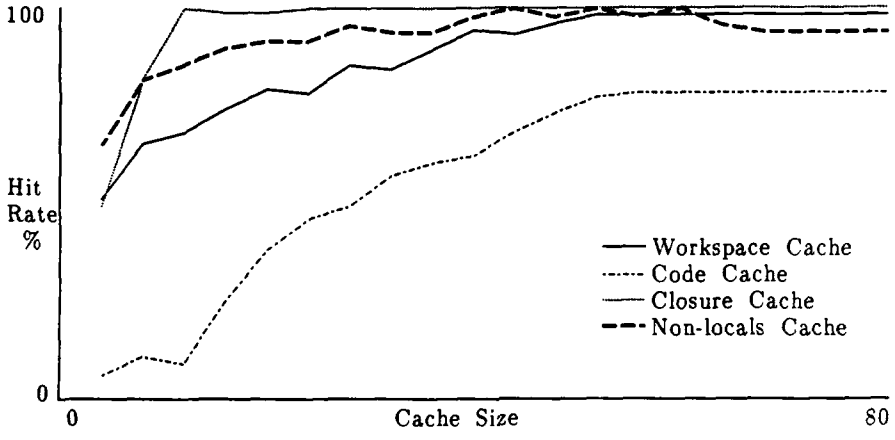


Figure 7.18. Write hit rate against cache size for program one.

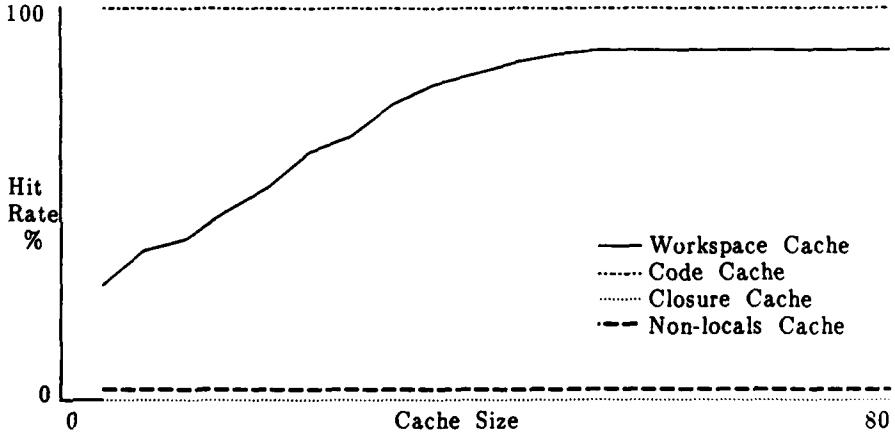


Figure 7.19. Read update rate against cache size for program one.

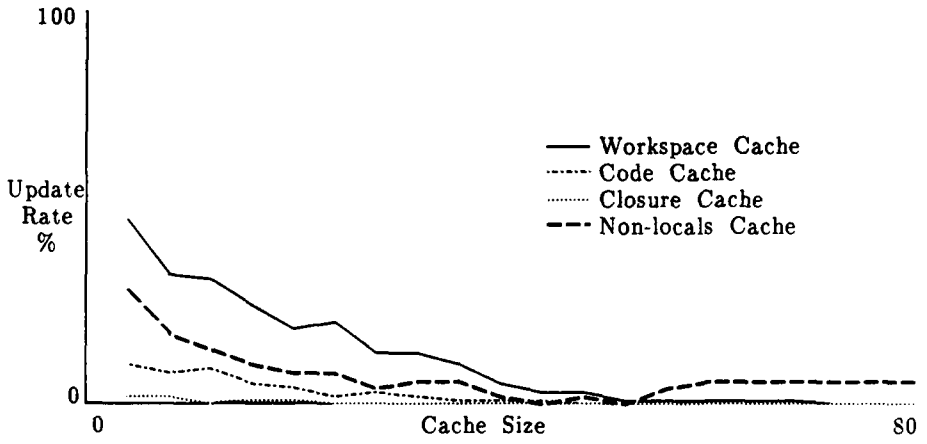
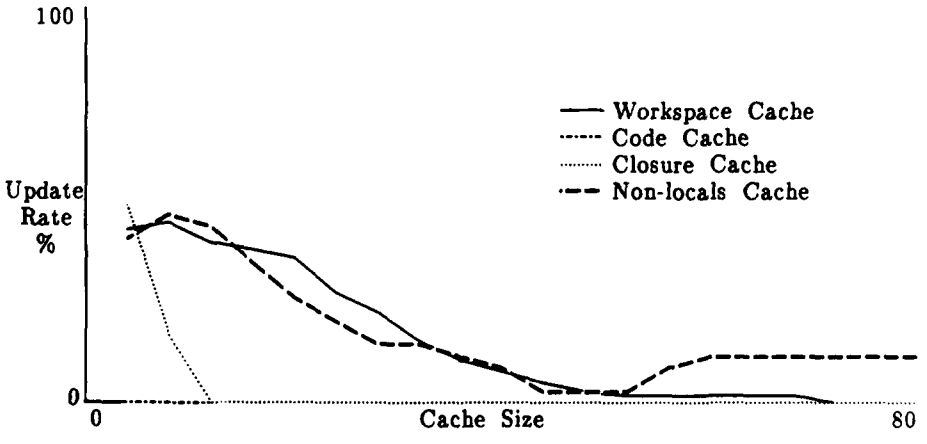


Figure 7.20. Write hit rate against cache size for program one.



The results shown in Figures 7.21-7.24 are from running a program which strips the blanks from a text, and creates blocks of words.

Figure 7.21. Read hit rate against cache size for program two.

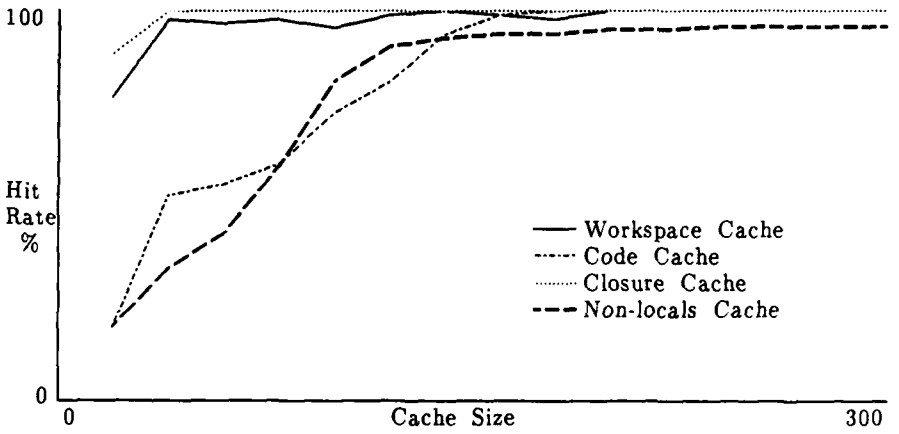


Figure 7.22. Write hit rate against cache size for program two.

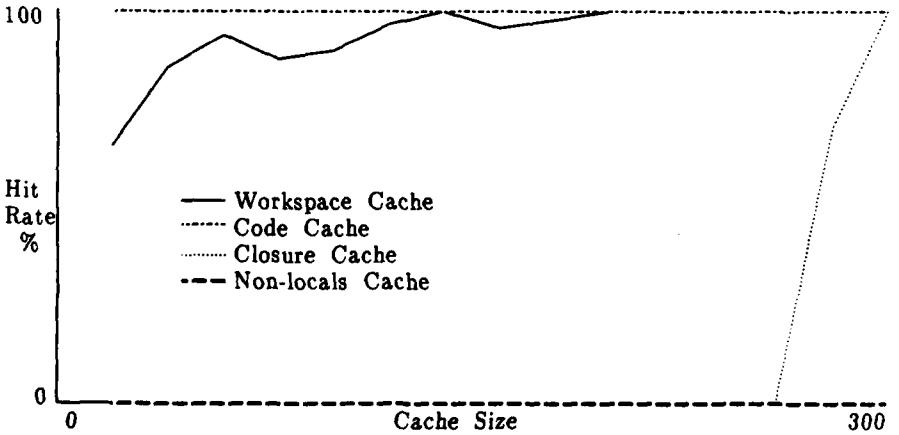


Figure 7.23. Read update rate against cache size for program two.

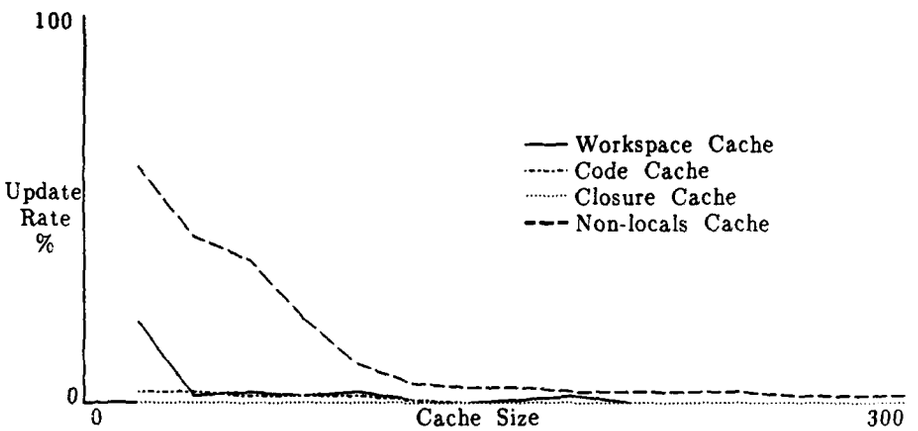
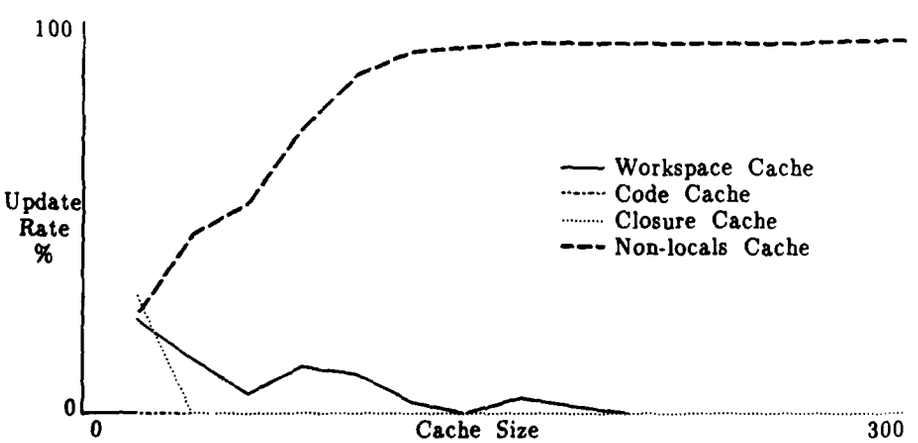


Figure 7.24. Write update rate against cache size for program two.



The results shown in Figures 7.25-7.28 are from running a program which uses a bubblesort algorithm to sort a list of words into alphabetical order.

Figure 7.25. Read hit rate against cache size for program three.

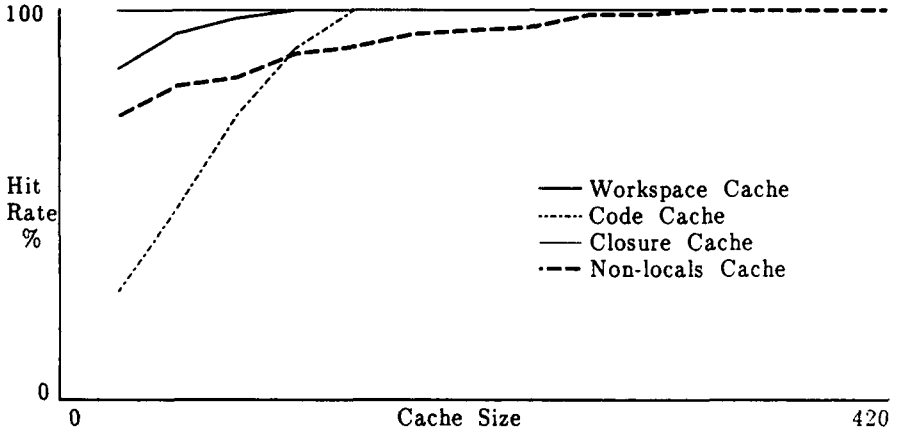


Figure 7.26. Write hit rate against cache size for program three.

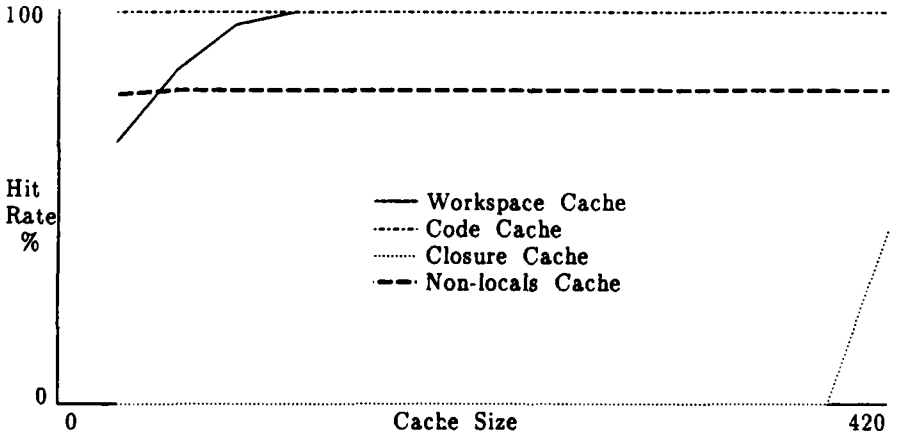


Figure 7.27. Read update rate against cache size for program three.

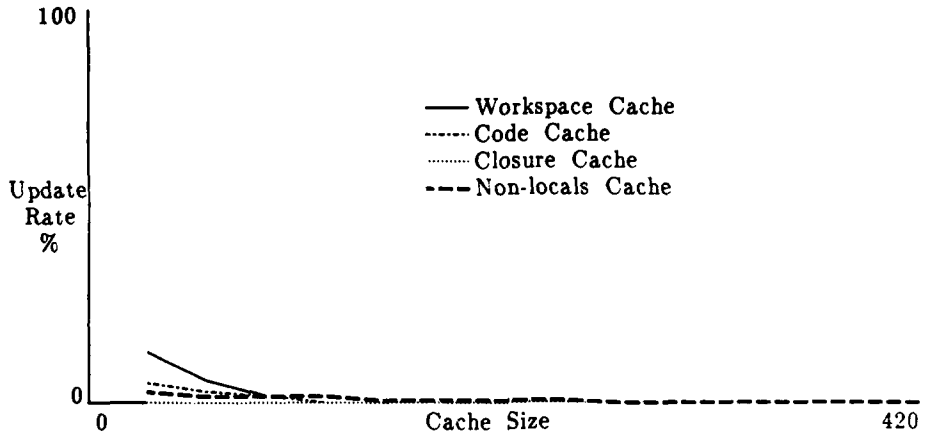
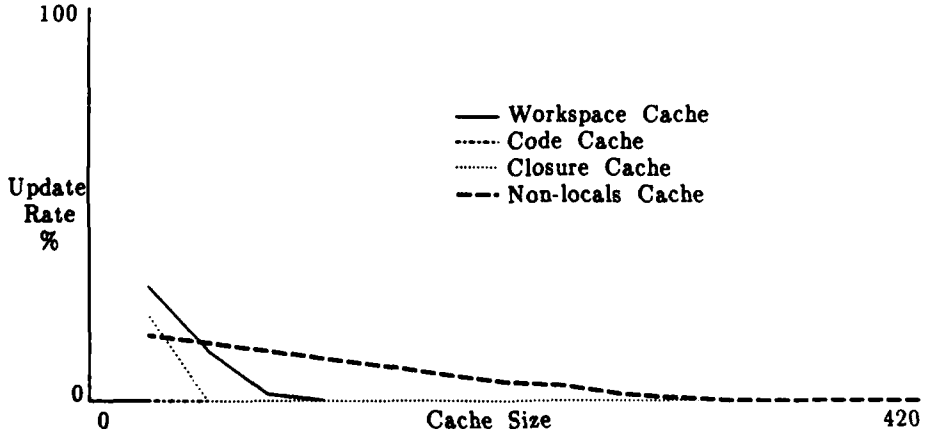


Figure 7.28. Write update rate against cache size for program three.



The results shown in Figures 7.29-7.32 are from running a recursive program which strips the blanks from a text, creates blocks of words and keeps a list of the stored blocks. .

Figure 7.29. Read hit rate against cache size for program four.

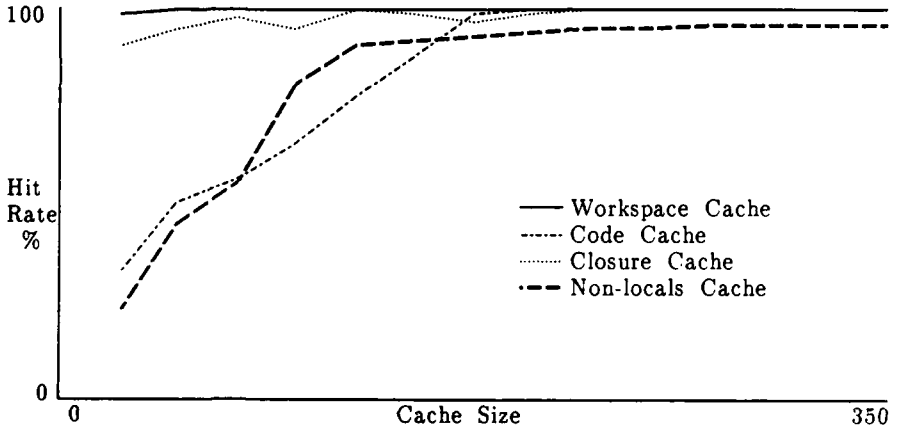


Figure 7.30. Write hit rate against cache size for program four.

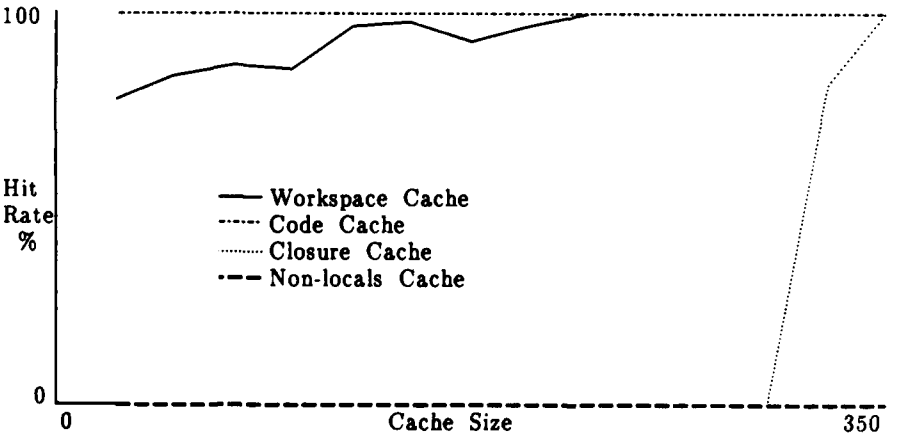


Figure 7.31. Read update rate against cache size for program four.

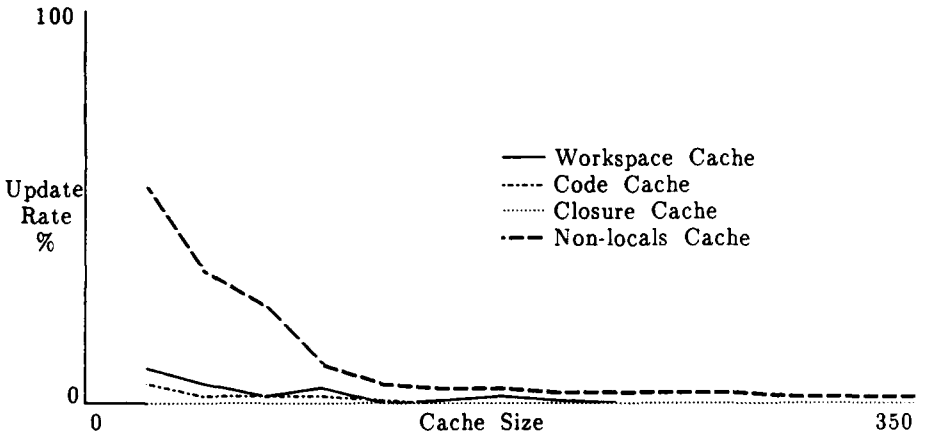
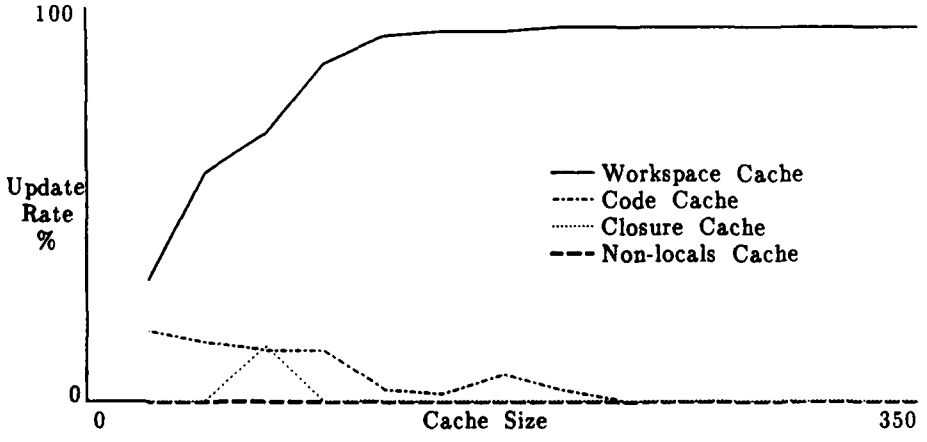


Figure 7.32. Write update rate against cache size for program four.



7.3 Performance vs Associativeness.

This series of tests are used to show the relationship between the degree of associativeness and performance of a single cache. The following graphs show the performance curves for a single cache running a program to strip blanks from a text, create blocks of word, and keep a list of stored blocks.

Figure 7.33. Read hit rate against degree of associativeness.

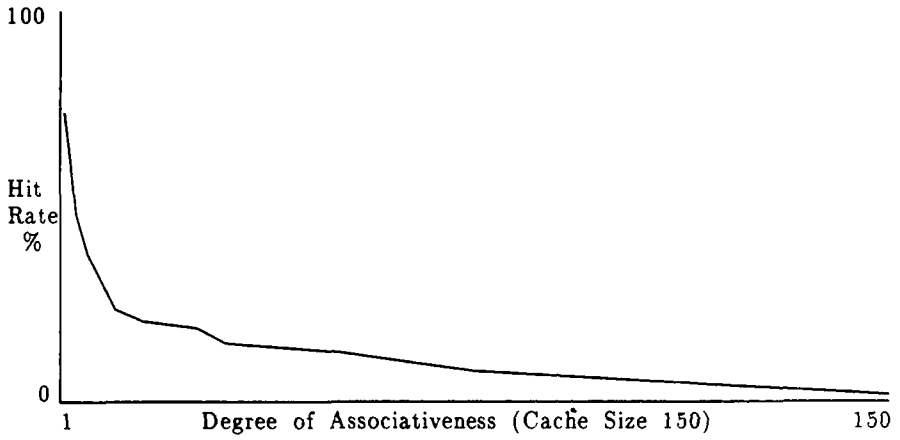


Figure 7.34. Write hit rate against degree of associativeness.

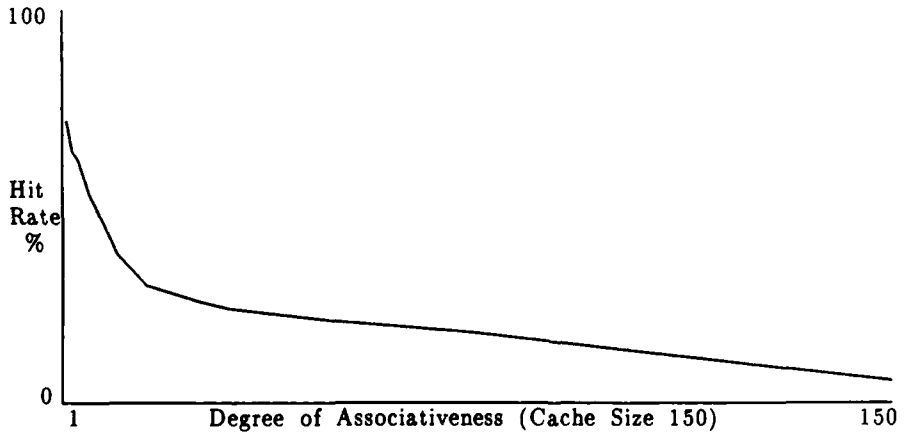


Figure 7.35. Read update rate against degree of associativeness.

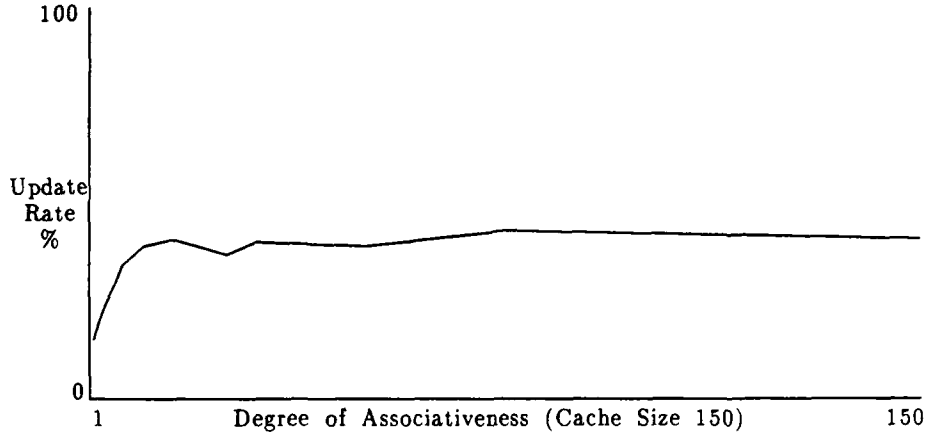
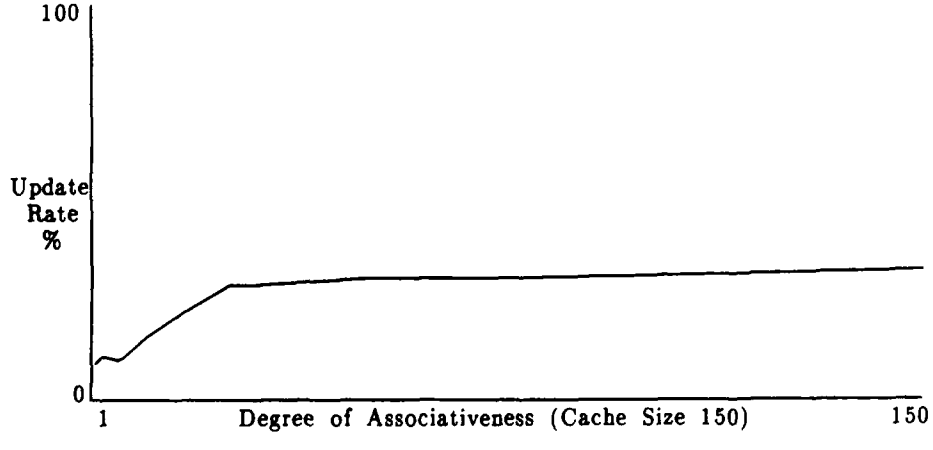


Figure 7.36. Write update rate against degree of associativeness.



8. Discussion.

8.1 The Effect Of Increasing Cache Size On Single Caches.

For a single cache, all the results show a similar trend - an increase in hit rate, and a decrease in update rate as cache size increases. In general, the results also present curves for hit rates which give indications of the characteristic curve proposed in the theory. Similarly, the update rate curves also suggest a curve similar to that expected. The results for the test program Towers of Hanoi, whilst still displaying the increase in hit rates and decrease in update rates, do not show a curve as expected. Instead, the results are much more linear in nature. Two explanations are possible, the first being that the range used in the test is insufficient to show the presence of anything other than a linear relationship (i.e the range of results lies between the toe and the knee of the curve). Taking further tests, with greater cache sizes (not shown) the trend is also linear. Thus that is not the correct explanation. Considering the nature of the test program, it is seen to be different to the other test programs. This difference provides a second explanation.

The Towers of Hanoi program, unlike the other test programs, is recursive. This means that the store is not accessed repeatedly - accesses are generally **single-shot** and subsequently discarded. Hence, the performance may only be improved by the method of brute force; a large increase in cache size is required to affect a significant increase in performance. Thus the ideal of a small cache greatly increasing the speed of the computer is unobtainable. However, for very small programs the efficiency may still become very high if the cache encompasses a large number of the locations used by the program. The program is then effectively running in cache memory rather than primary memory.

The other programs are all iterative, and hence should give rise to results which should portray the expected curve rather more closely than those of the Towers of Hanoi program. Initially a sharp increase in hit rate should be observed. This sharp initial rise in hit rate is indeed shown by all the remaining programs. This is due to the fact that, although the increase in size is small, this increase represents a large proportion of the original cache size. Considering update rates, a similar sharp decrease is noted, for similar reasons, although much less prominent. This is due to the small update rates involved which mean that the rate becomes very small very quickly, and so any changes are also necessarily small. As expected, the rate of reduction in update rate decreases with an increase in cache size.

Increase the size of the cache still further, and these effects diminish. The increase in hit rate is reduced, ultimately towards zero (although it is likely that the cache will encompass program store requirements before this point is reached). This is because the rate of increase in cache size decreases, and there is less data to cache, which is not cached already. A clear example of this effect is found in the write hit rates for program three. As predicted, the rate of all the hits in this region (saturation) is less than 100%.

The toe of the curve is not represented in any of the results. This effect will only appear when using very small cache sizes, when a large number of primary locations are mapped onto each element in the cache. However, the cache size required for this effect to become prominent is so small that it is extremely unlikely that the effect would ever be observed in normal sized programs. In the short test programs used it is almost certain that this will not be observed. This effect is not observed on even the larger test programs such as frag, even when using a cache consisting of a few tens of elements. Thus investigation of this area of the graph is difficult to say the least. It is of no practical benefit anyway, as a cache size of the sizes mentioned would not be worthwhile incorporating into a practical system, since the size of commercial memory units (of the speed required) is greater than the sizes envisaged.

Although these results generally follow the characteristic curve, some deviation from point to point is observed. These deviations are dependant upon both program and policy and are usually caused by the periodic nature of the memory accesses. This is best described by example.

Consider a cache of ten elements, using a direct mapping scheme. A program accesses locations one and eleven alternately. Both locations are mapped onto element one in the cache. A reference to location one will delete the still active data from location eleven and vice versa, thus causing a low hit rate. Altering the number of elements in the cache, even reducing its size to nine elements improves performance greatly.

The periodicity of the program will affect different cache structures in different ways. A direct mapping cache will be the most affected. Associative caches may also be affected, but to a lesser extent as the degree of associativeness increases.

This '**resonance**' may be beneficial and create a more even spread of entries throughout the cache, and thus increase hit rate. However, it is more likely to be noticed as a concentration of accesses in a particular region within the cache, causing far greater numbers of deletions of active data than normally expected, thereby reducing hit rate. These disparities are unavoidable, although often the difference of a single element will produce a marked settling effect. These resonant points may have harmonics, but none are apparent from the results, which are necessarily coarse and cover but a small range of cache sizes.

8.2 The Effect Of Increasing Cache Size On Discrete Caches.

The results obtained are for a range of cache sizes identical to those used in the examination of the single cache to enable a comparison of the efficiency of the two types. However, this presents a problem in discussing the individual merits of increasing the size of a discrete cache. This is because the blocks of data cached by the individual caches is much smaller than that of the single cache. Hence the hit rates are high and the update rates low - even for the smallest cache sizes investigated. In fact in several cases, the rates are 100% and 0% respectively. Thus comment upon the effect of increasing the size of these caches is often nonsensical. However, it may be said that the caches in question need only be very small to encompass the working set.

The effect of increasing cache size upon the other caches illustrates the theoretical curve more clearly than the results for the single cache. The curves are generally much smoother also. These effects are due to the separation of the data, which prevents interference between types. Effects due to the periodicity of programs are also reduced. This may be due to references of a particular type being sequential, as opposed to having a periodic nature which may arise in the whole program, for example moving data between blocks.

In some cases, the results obtained are totally contradictory and unexpected. The results in question are found in the graphs of write update rates for the programs word and frag. The update rates increase with cache size. It occurs in the non-locals cache of the program word. In frag, the fragmentation is increased, and the effect becomes noticeable in the workspace cache. It is likely that these effects occur because the programs amend the data in situ. Thus a miss is likely to require an update.

8.3 The Effect of Policy.

It may be seen that different policies will be affected in different ways. The random policy is largely unaffected by any periodic nature of the data, whereas this has a noticeable effect upon the direct mapping cache. The random policy is also little affected by any single-shot accesses. The LRU and LRC policies are significantly affected by this. Hence, the affect of a program can be summarised as follows.

A program with periodicity in its accesses (e.g locations 1, 11, 21 in a cache of size 10) will affect direct mapping caches. A program with many single-shot accesses will affect LRU and LRC policies.

It should also be noted that the random number generator is not entirely random, and so resonance type effects may be induced in a random set-associative cache using this policy. This is irrespective of any periodic nature within the program, but is a type of periodicity dependant upon the interaction of the two.

Obviously, all types of cache will be affected by the type of program, but this will be to a greater or lesser degree depending upon its structure and the program involved.

8.4 The Effect Of Calling Blocks On A Single Cache.

Comparing the results for a single cache, the difference in the efficiency of policies is great. A direct mapping cache is seen to be as effective as a set associative cache using the random displacement policy. However, the LRU and LRC policies are markedly less effective, both being similar. This is unexpected, as these policies should improve on the other, less complex policies. They are less efficient because of the operation of the Flex like instruction set. Each block has several **overhead** words associated with it. These words comprise of some status flags, the size and type of the block, and some procedure call link words. These words need only be accessed once when the block is called. If the overhead words are cached they will occupy entries which could be used to store active data, and thus reduce the performance of the cache.

The overhead words are invariably misses when written to store, and thus data already in the cache is discarded. The discarded data is likely to be used in the near future, whereas the overhead words are not.

In a direct mapping cache, these overhead words force the same cache entry to be discarded repeatedly when the block is initialised. Thus the efficiency is reduced, the entry being, to a large extent, redundant.

However, in a fully associative cache the effect is much wider ranging. Consider the fully associative cache, as the same principles apply to a set associative cache, but to a lesser degree. The overhead words not only cause the least recently used/cached entries to be discarded, but become themselves the most recent entries. Hence they will be the last entries in the cache to be discarded. This means that they are likely to remain in the cache longer than in a direct mapping cache, where they may be deleted with the next memory access. The cache, therefore, has reduntant entries for a longer period of time. In addition to this, the older entries in the cache will be discarded earlier than if the overhead words were not present. Thus, entries which would normally be resurrected at the last moment, will be discarded, further reducing the efficiency of these policies.

The solution to the problem is not to cache the overhead words, but access main store. This is only possible because of the blocks in a capability computer.

8.5 Cache Structure - Effect of Associativeness.

Using the simulator to show the effect of structure upon performance is difficult to accomplish, as it is very slow for large degrees of associativeness. A small cache must be used so as not to encompass the working set if any attempt is to be made to obtain a valid result. Degrees of associativeness investigated are necessarily restricted by the cache size, and a regular increment is impossible. Hence, resolution in some areas of the results is poor.

From the graphs, it may be seen that the effect of associativeness is not as expected (as shown by Figure 3.3). In fact, the fully associative cache fares much worse than both the direct mapping cache and the set-associative caches. This result is explained by the overhead words. For reasons explained above, the results show the increased effect of these words as the degree of associativeness increases.

8.6 The Effect Of Discrete caches.

The results obtained by using discrete caches show a clear improvement over the single cache. Take, for example, the read hit rate for Hanoi. Discrete caches with sizes 8, 8, 12, and 44 (a total of 72) offer a several percent improvement over a single cache size of 80. Even using four caches of the same size can offer improvements, although it may be seen that this is wasteful. The closure cache need only be very small in the majority of cases, a size of as little as 8 is often sufficient, and should not require many more entries, even for large programs. The non-locals cache may also be reasonably small, but considerably larger than that for closures. The workspace and code caches must be reasonably large. These proportions reflect the sizes of the parts of an 'average' program (if there is one!).

8.7 The Relationship Between Sizes Of The Discrete Caches.

To profit most from the use of discrete caches, some consideration should be given to the relationship between the sizes of the individual caches. The results indicate that the size of the workspace cache should be the largest, with a smaller code cache. The closure and non-local caches should be much smaller, and of approximately equal size.

8.8 The Advantages Of The Writeback Scheme.

The update scheme of writeback is the more useful scheme because it reduces delays, both in writing to main store, and by avoiding conflicts with DMA accesses. The problem of the main store not being consistent is serious, but the updates can be done in parallel to accessing the cache. The implementation is slightly more complex than that for writethrough, but the possible speed advantages make it worthwhile.

8.9 Practical Considerations.

As in many practical studies, the results appear to bear little resemblance to those expected. Considering that the test programs only addressed a range chosen to give the most interesting and useful results, the results show but a small part of any ideal theory. With this reservation, it may be said that the results are represented by the theories proposed. Thus they are an aid to cache design, reducing recourse to practical tests, which cannot be comprehensive and may, at times, be misleading. This is because the results show the effect of specific programs, and are sometimes quite different to the overall processing in a machine.

In considering the efficiency for a practical system, not only must the performance of the algorithms be considered, but also the cost of implementing them on a real machine. The various structures not only affect performance, but also affect the cost of development and hardware. A method of analysing the performance benefits against cost and complexity of design would be helpful, but the results obtained in this study suggest that the tweaking of performance is not worthwhile at the present time. Even when the cost of the hardware falls, it is likely to be the high-speed memory, rather than the controlling logic, which enjoys the greatest reduction. Hence it seems that, even in the future, the direct mapping cache will remain the most viable implementation.

8.10 Unfair Comparisons.

In this discussion, comparisons have been made between caches containing the same number of data elements. However, as noted in the description of cache structures (section 2.3), a fully associative cache will require more redundant memory in which to store the tag than a direct mapping cache. This memory has to be as fast as the data section of the cache, for obvious reasons, and could perhaps be better utilised as data store in a simpler cache. The wastage involved in a fully associative cache is proportional to the size of the main store. This is due to the number of bits required to store the address. The size of the data word remains constant irrespective of the structure of the cache.

8.11 Further Investigations.

The initial investigations have yielded some interesting results, but have also raised further questions worthy of study. Particular areas worthy of consideration are:-

a) The structural difference of caches (taking a ten element cache as an example): Which is the more efficient arrangement, a two-way associative cache of five lines, or a five-way associative cache of two lines?

b) As the size of the cache increases, the advantages of a fully associative structure over a direct mapping structure should become diminished, until the resulting gap in efficiencies becomes negligible. At what point can no practical benefit be gained from the fully associative cache?

c) When comparing caches, differing either in policy or structure, conflicting results often occur depending on the cache size range examined. A method of deciding the range to be investigated is worthy of consideration, reducing the volume of misleading data, and reducing the load on computing services.

9. Conclusions.

On a conventional computer, the most efficient structure for a single cache is a fully associative cache. The set-associative cache is less efficient, but yields a better performance than the direct mapping cache. There is, however, little difference between the performances of these structures.

On a capability computer, the efficiency of the structures is affected by the overhead words. If these are not cached, then the fully associative cache is the most efficient, followed by the set-associative and the direct mapping cache. However, if the block data words are cached, the direct mapping cache is most efficient, and the order of efficiency is reversed.

The capability architecture allows the block types to be exploited, as the block type is always known. This enables discrete caches to be used to cache specific block types. The performance of a single set-associative cache may then be obtained (or improved upon) by using several direct mapping caches with a total size no greater than that of the single cache. Thus, the performance benefits of the set-associative cache is available with the simplicity of the direct mapping cache.

Storing overhead words in the cache results in a decrease in performance for any of the cache structures, particularly the associative caches. To prevent this, the overhead words should not be stored in a cache. It would be possible to store the overhead words in a separate cache, but it is unlikely to be of much use. This is because they do not exhibit locality. often being used once and then discarded. Thus the cache will have a poor performance, and would be better utilised to cache an extra block type which exhibits locality.

The writeback scheme is preferable to writethrough. Writeback is more complex to implement, but not greatly so. The extra logic required to determine when an entry should be updated to main store is worthwhile. This is due to the speed improvements gained because main store is only addressed when necessary. This also means that DMA conflicts are reduced considerably.

Discrete caches should be implemented with care. Although they provide an improvement over a single cache, careful choice of the type of blocks cached, and the size of each cache is required to obtain the best performance from them. Caches for code and workspace blocks should be large, whilst those for closure blocks should be quite small. The caches should be used for block types displaying properties of locality rather than those that do not.

10. References.

"Operating Systems Theory."
E G Coffman and P J Denning
Prentice-Hall 1973

"Flex Firmware."
I F Currie, P W Edwards and J M Foster
RSRE Report No. 81009 September 1981

"An Introduction To Operating Systems." Revised First Edition
H M Deitel
Addison-Wesley Publishing Company 1984 ISBN 0-201-14502-2

"An Introduction To The Flex Computer System."
J M Foster, C I Moir, I F Currie, J A McDermid, P W Edwards,
J D Morison and C H Pygott
RSRE Report No. 79016 October 1979

"Flex:A Working Computer with an Architecture Based on Procedure values."
J M Foster, I F Currie and P W Edwards
RSRE Memo No. 3500 July 1982

"A simulator to Investigate Cache Policies for a Flex Type Instruction Set."
J G Haines
Internal Report June 1989

"A Study of Store Management Policies for Incremental Garbage Collectors."
C L Harrold
RSRE Report No. 86018 December 1986

"Cache Memories."
A J Smith
ACM Computing Surveys Vol. 14 No. 3 pp. 473-530 September 1982

"Cache Operations by MRU Change."
K So and R N Rechtschaffen
IEEE Transactions on Computers Vol.37 pp. 700-709 No. 6 June 88

"A Class of Compatible Cache Consistency Controls and Their Support
by The IEEE futurebus"
P Sweazey and A J Smith
Proc. Of The 13th International Symposium on Computer Architecture 1986
pp 414-423

"Protection and Security Mechanisms In The SMITE Capability Computer."
S R Wiseman
RSRE Memo No. 4117 January 1988

"The SMITE Computer Architecture"
S R Wiseman and H S Field-Richards
RSRE Memo No. 4126 January 1988

"Benchmark Synthesis Using the LRU Cache Hit Function."
W S Wong and R J T Morris
IEEE Transactions on Computers Vol.37 No. 6 June 1988 pp. 637-645

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	REPORT 89015		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS AND RADAR ESTABLISHMENT ST ANDREWS ROAD, MALVERN, WORCS WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

A STUDY OF CACHING IN CAPABILITY COMPUTERS

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
HAINES, J G			09.1989	44

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This report describes an investigation into the use of cache memories in capability architectures. A theory to describe the relationship between the size of cache and its efficiency is proposed. The use of a simulator to test the theory and compare various cache management policies is described, and the results obtained are discussed. Conclusions are drawn as to the best caching strategy for the SMITE capability computer.